

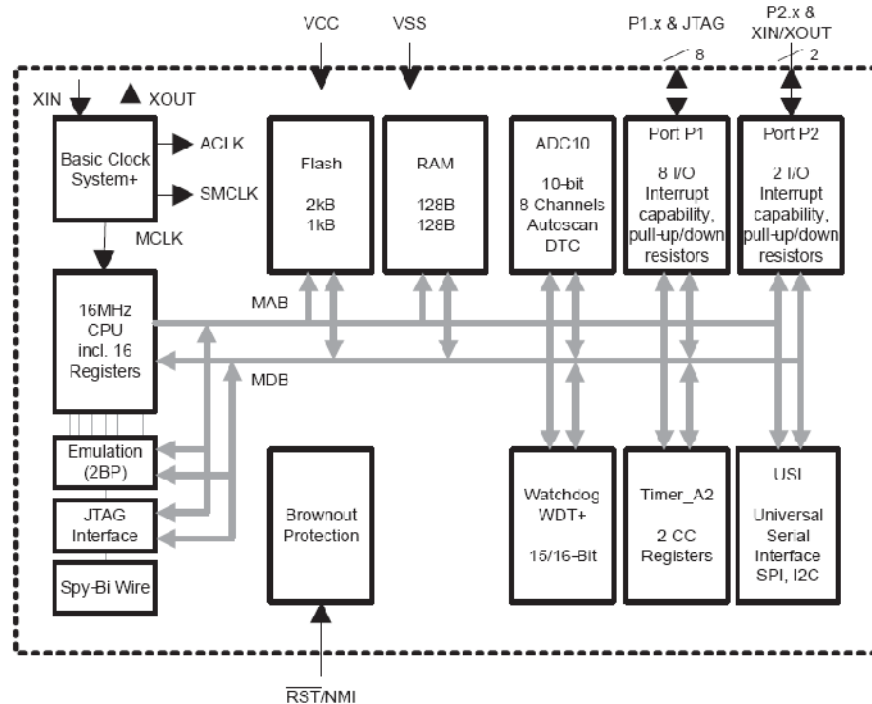
University of California Berkeley
Department of Electrical Engineering and Computer Sciences
 EECS 100, Professor Bernhard Boser

LABORATORY 9 v3
 MICROCONTROLLER

Microcontrollers are very much slimmed down computers. No disks, no virtual memory, no operating system. Think of them just like other circuit components with the added benefit of being configurable with a program. Because of this microcontrollers can be coaxed to do all sorts of things simply that otherwise would require a large number of parts. Simple microcontrollers cost less than a dollar and hence can be used in almost any project. Indeed they can be found in toys, electric tooth brushes, appliances, cars, phones, electronic keys, you name it.

Being programmable also means that they must be programmed. In EE100 focus on the electrical interface of microcontrollers and their use as electronic components. The programs we use are very simple and consist to a large part of pasting snippets of code together. In fact, much like checking the application notes of electronic components for circuits that do what we need, it's always a good idea to search the web for code that performs the job we need or is at least a good starting point. Most of the code snippets shown in these lab guides are copies of code from the manufacturer's website. Feel free to improve on the example programs.

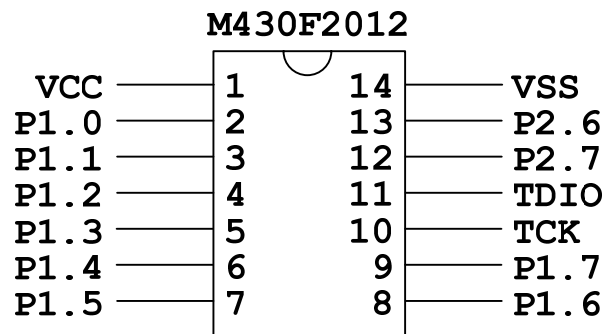
At least half a dozen simple microcontrollers are available from several manufacturers, all with their own advantages (and quirks). In this course we use the MSP430 from Texas Instruments whose strength are low power dissipation and a regular instruction set. Neither feature is critical for EE100 and we could use another part just as well.



The diagram above shows the architecture of the MSP430 (specifically the model MSP430F2012). “CPU” is the part that actually performs computations (e.g. additions). Note that microcontrollers usually lack hardware for multiplication or division. These operations can be emulated in software, albeit at the price of low execution speed. The clock system sets the operating speed (16MHz maximum for the controller we are using, compare this to 2GHz or so for present day laptops). A 555-timer like clock is built right into the chip; alternatively an external oscillator can be used if higher precision is required.

“Flash” is a nonvolatile memory for storing programs and configuration data. “RAM” is where temporary variables go. Note again the contrast to full blown computers: microcontroller memory is typically a few kBytes flash and a few hundred Bytes RAM. Most laptops today have at least a GByte RAM, a million kBytes. You don’t need this in an electrical tooth brush. “JTAG” and “Spy-Bi Wire” is a nifty interface for programming and debugging (it’s got no keyboard or LCD display). We will use this interface to talk to the controller though USB from a desktop computer. This of course is used only for development, once completed the controller works standalone from the program stored in Flash memory.

The really interesting parts are the peripherals. Ports P1 and P2 are digital I/Os that can be configured as inputs or outputs. They can be used for simple I/O with switches or LEDs; in later labs we will see much more sophisticated uses of this simple interface. The other block we will use is the “ADC10”, a 10-bit analog-to-digital converter that serves as a bridge between the usually analog “real” world and the microcontroller. For example we can use it to interface the strain gage circuit designed in an earlier lab to the microcontroller and make a full blown (simple) balance with display out of the combination!

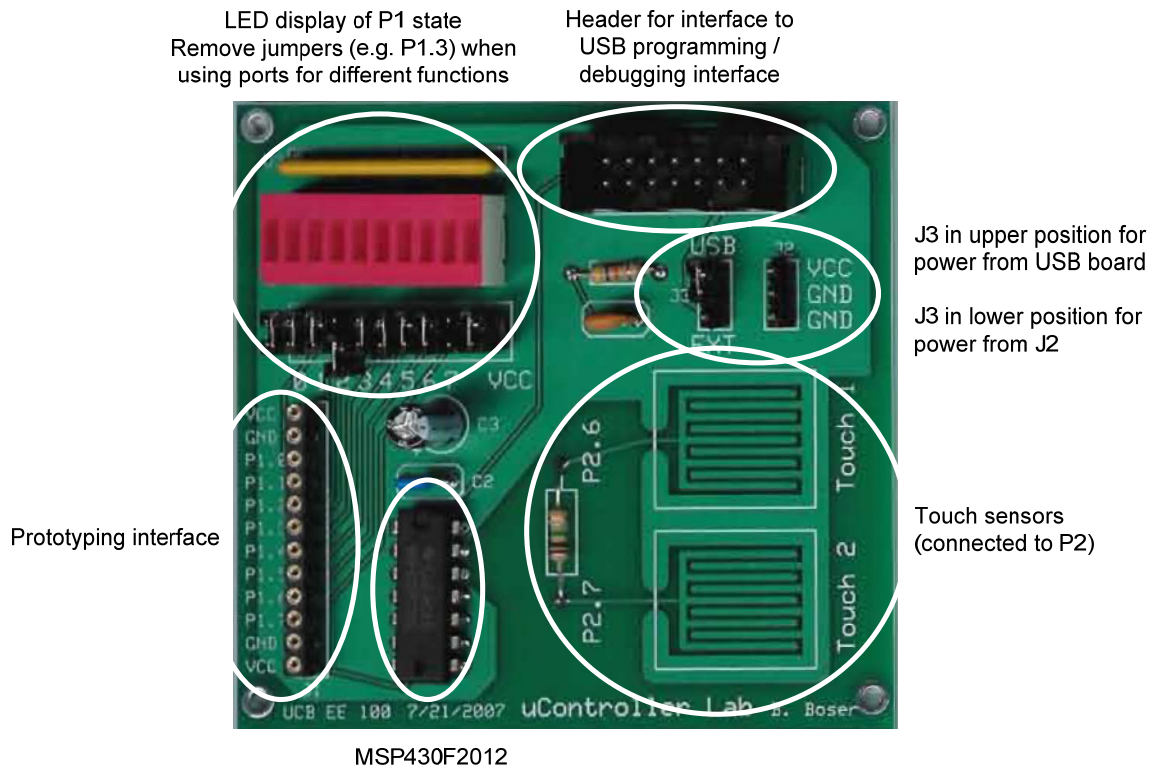


The diagram above shows the pinout of the MSP430F2012. It has only 14 pins for power (VCC) and ground (VSS), the debug interface (TDIO and TCK) and ports P1 and P2. Looking carefully you will observe that there are eight pins for P1 but only two for P2. The other ones don’t fit with a 14 pin package. There also are no separate pins for the ADC. Instead, digital IO pins can be reprogrammed as ADC inputs as needed. Several dozen MSP430 microcontrollers are available with their main difference being the number of pins and the amount of memory. This permits you to start with a small model, and as the project grows move to models with more features without having to change the programs developed for the smaller parts.

With only 14 pins we certainly could wire up the microcontroller on a protoboard. The custom board shown below makes wiring even simpler. It also features LEDs for debugging and a touch interface which we will use in a later laboratory.

In addition to the microcontroller, the board features a header for interfacing with a ribbon cable to the USB interface which is also supplying power to the board. Three capacitors and a resistor are used for filtering power and resetting the device after power is applied. These devices are specified by datasheet of the device. A header strip on the left side of the board exposes P1 and P2 and power for prototyping.

Alternatively P1 also can be connected to onboard LEDs through jumpers. This of course makes sense only for pins that are configured as digital outputs. Removing a jumper (or setting it on a single pin as shown for P1.3 for not losing it) makes that port available for other functions such as digital input or the ADC. The rightmost position (VCC) is for monitoring the power supply and always on when power is supplied to the board and the jumper is inserted. Jumper the upper two terminals of J3 as shown to enable power from the USB interface.



LAB REPORT

Lab Session:

Name 1:

SID:

Name 2:

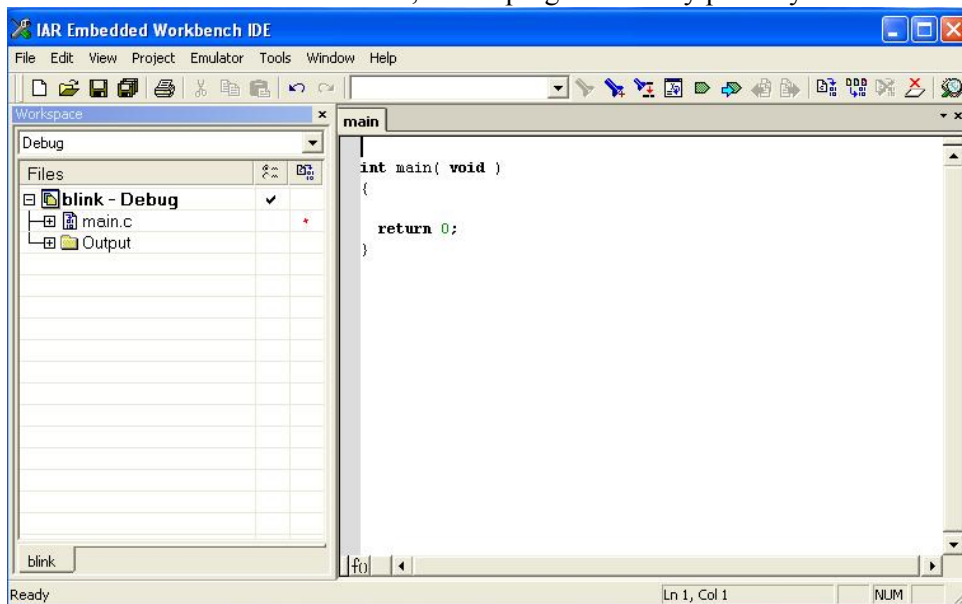
SID:

You will have to read through all of this to do the prelab in part 3!

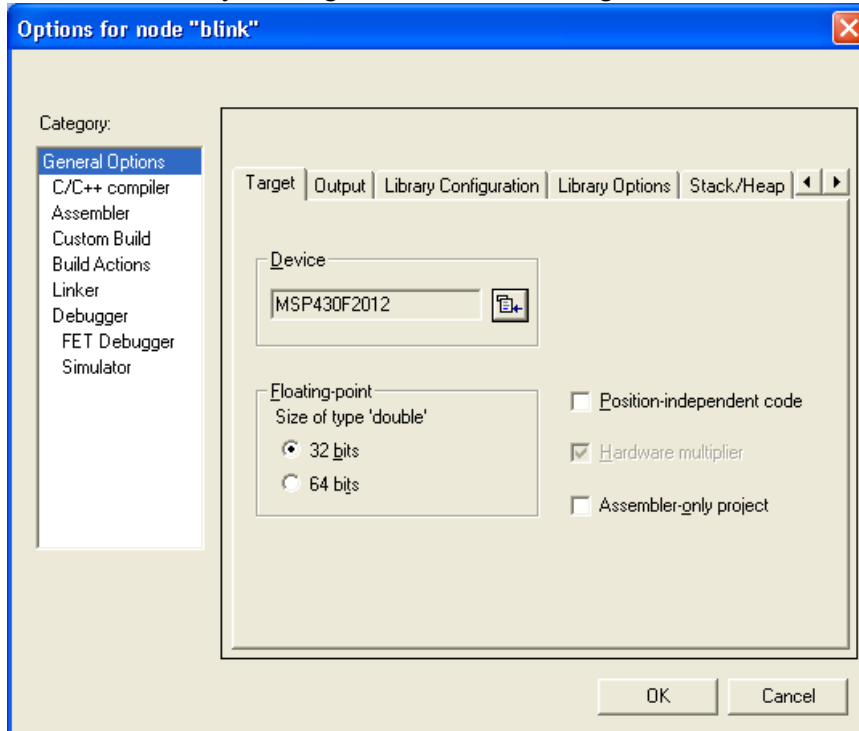
1. Blinking light

In this laboratory we will familiarize ourselves with the EE100 uController board and the MSP430 development tool. We will start with writing the notorious “Hello World”, which for a uController is a blinking light.

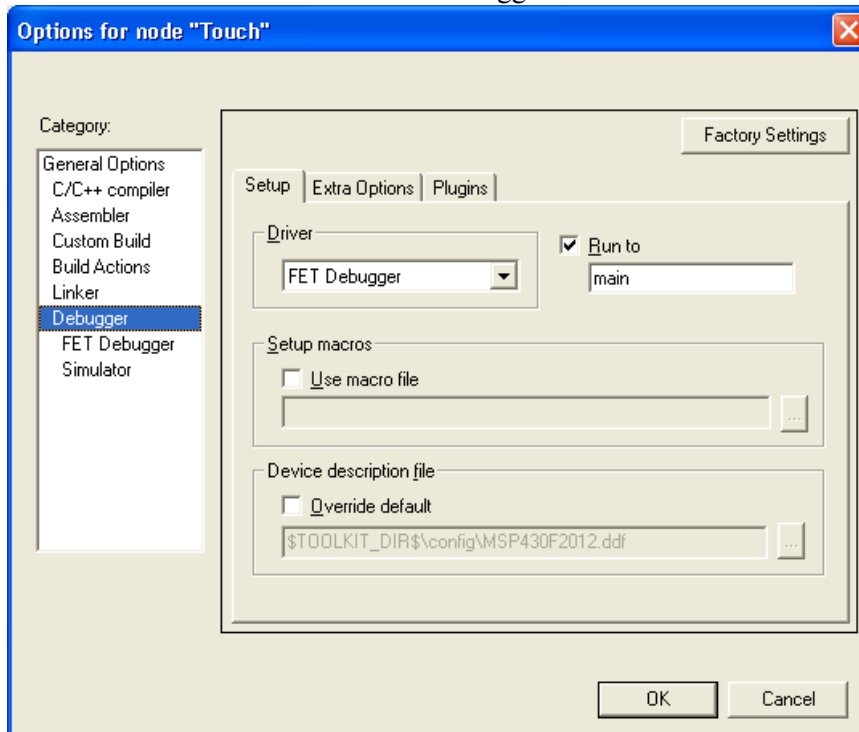
1. Connect the EE100 uController PCB to the desktop computer with the ribbon cable and the MSP430 USB-Debug-Interface (MSP-FET430UIF). Use a standard USB cable to connect the debug interface to a computer with the “IAR Embedded Workbench IDE” software. This software is installed on the computers in the EE100 laboratory. Alternatively you can download it from the TI website and install on your own computer if you prefer.
2. Start the “IAR Embedded Workbench IDE”. Choose “Create new project in current workspace”. A dialog with options appears. We will write our program in the “C” language. Expand that choice, click on “main” and then click “OK”. The program asks you to name your project. Call it “blink”. Hit enter.
3. Your screen looks as shown below, with a program already partially written.



4. Before finishing our program we need to configure the tool for the MSP430F2012. Choose the menu “Project→Options ...” and click on the “General Options” tab. Set “Device” as shown in the screen below by clicking on the button to the right of the field and navigating the choices.

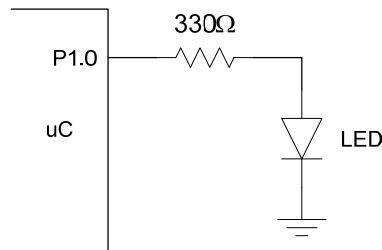


5. Still in the options dialog, verify under the “Debugger” tab that driver is set to “FET Debugger”. Also make sure that under the “FET Debugger” tab “Connection” is set to “TI USB FET”.



6. Still in the options dialog, click the FET Debugger option on the left and choose TI USB FET under the connection selection.
7. Click “OK”.

Put a jumper into position 0 of J5 to connect the microcontroller port P1.0 to the leftmost LED. A jumper is simply a removable connector that is used to selectively connect two neighboring pins from a circuit board. When the jumper is placed on the two pins, they are connected by a low resistance path (like a wire); when the jumper is not placed, the two pins are electrically unconnected. With the jumper in place, the circuit on the board is as follows:



The resistor is needed since LEDs behave like diodes with very large current flowing for voltages above a threshold (around 1V for LEDs). The microcontroller changes between 0V and 3V (for logic low and high) and can burn out when connected to an LED without a series resistor.

Place a jumper also in the VCC position of J5 to verify that power is supplied to the board when you start debugging.

Before we can write a program to do what we want, we need to add a couple of lines of code that will enable proper operation.

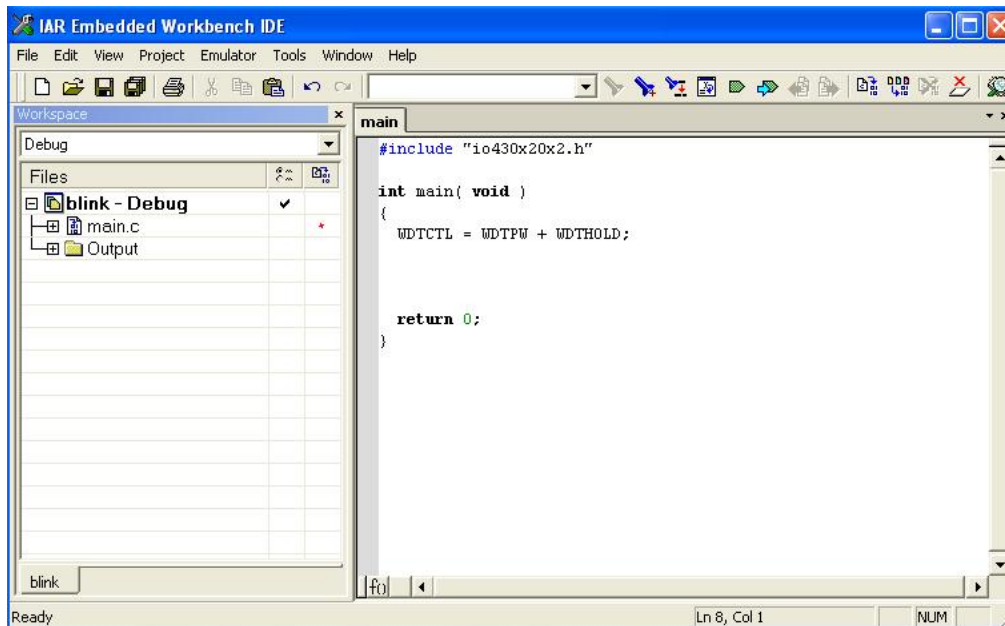
1. Add the required header file as the first line in the program. The header file contains code that is specific to the particular chip you’re using.

```
#include "io430x20x2.h"
```

2. Add a line to disable the watchdog timer. This timer is used to reset the processor after a specified time, and we don’t need this feature. Add this as the first line inside the brackets of your main function.

```
//Stop watchdog timer to prevent time out reset  
WDCTL = WDTPW + WDTHOLD;
```

Your code should look like this:



Now we will write the program that turns P1.0 on and off.

3. First we to configure P1.0 as a digital output using the following instructions:

```
P1OUT &= ~BIT0;
P1SEL &= ~BIT0;
P1DIR |= BIT0;
```

The first statement sets the output to zero. It is not strictly needed here since we do not care if the LED is on or off when the microcontroller starts. However it is a good idea to always first set an output to a known state before enabling it. Imagine we were programming a rocket ship ...

The next two statements configure P1.0 as digital IO with direction set to output. As you would expect, the same statements with BIT1 substituted for BIT0 enable P1.1 as output.

Put these statements just after the code that disables the watchdog timer (don't worry about that timer; just make sure the code is there to turn it off).

4. The following statements set the output low or high, or toggle its state:

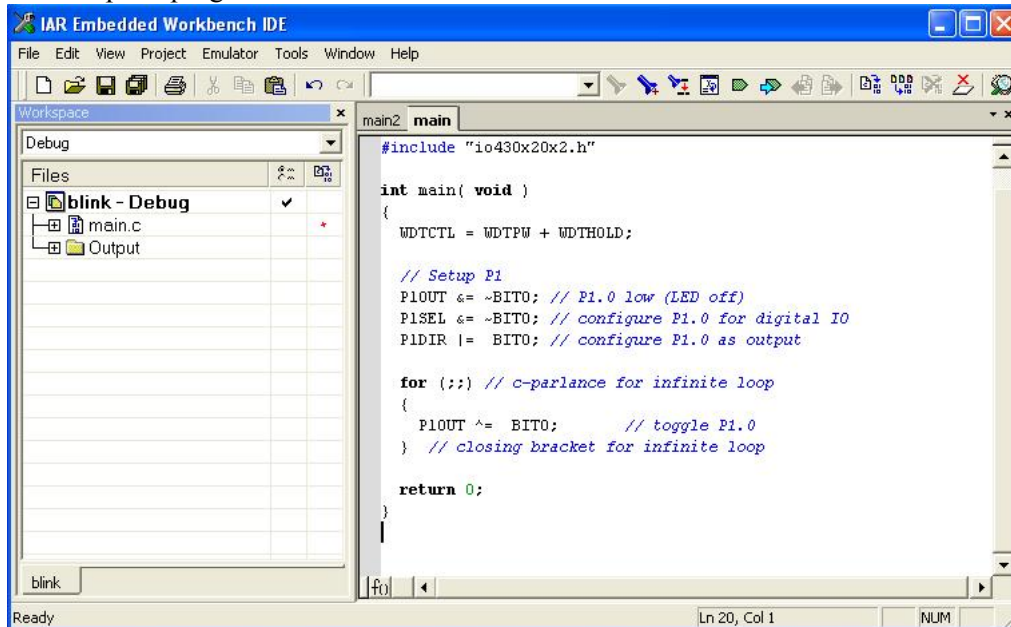
```
P1OUT &= ~BIT0;    // P1.0 low
P1OUT |= BIT0;     // P1.0 high
P1OUT ^= BIT0;     // toggle P1.0
```


Text after // is treated as comment and there only for documentation. If you have ever taken a programming course you know that we are supposed to document our code but few of us actually do it. Join the few.





5. The toggle version of the above statements is appropriate for blinking a light. Since we want to continuously toggle the light, we enclose the statement in a loop:

```
for (;;) { // c-parlance for infinite loop
    P1OUT ^= BIT0; // toggle P1.0
} // closing bracket for infinite loop
```

6. The complete program looks as follows:



7. Compile it by clicking on the right most button  in the toolbar. You may need to provide a workspace name. Carefully examine possible error messages (or ignore them and waste lots of time in frustration). A new window pops up.

Click the  button (2nd row in the toolbars). When done you can stop the program by clicking  or modify the code and click  to recompile and restart with .

8. If everything went right the light turns on but does not blink. At best it is a little dimmer than the power LED.

The reason is that the microcontroller turns the LED on and off a few million times per second ... too fast for our eyes to follow (at least mine, tell me if you can see the blink).

9. The simplest fix is to give the microcontroller a bit of extra work to slow it down. Microcontrollers are not students and hence do not mind. Since we will often have need for such delays, we package this function in a subroutine that we can easily reuse in other programs. Here is the code:

```
void delay(unsigned int n) {
    while (n > 0) n--;
}
```

Once we have added the above code to create the delay subroutine, we can call this subroutine with the following statement (keeping in mind that we cannot call the subroutine before it is created and thus need to create the subroutine above our main routine):

```
delay(30000);
```

This statement causes the microprocessor to spin 30,000 times in a loop decrementing the variable n. Note that there are many ways that subroutines can be defined. Feel free to use another method or ask your GSI if interested to know more.

10. Add this delay line into your for loop so that you can see the blinking. Your code should now be running an infinite loop consisting of decrementing n many times (wasting time) and then toggling the state of the light.
11. You can change the rate of the blinking by varying the argument to delay. Beware: the maximum permitted value is a little over 65'000 (you'll get a warning if you exceed the maximum which you are free to ignore if you need a few hours of frustration with stuff that doesn't work). For longer delays you can e.g. use several delay statements. Can you get the LED to blink at a rate of 1Hz?

Ask the lab TA to verify that the light is blinking: _____ of 10 M

2. Bar Graph

Now we will program a function `void bar(int n)` that turns on LEDs 0 ... n-1. E.g. the call `bar(3)` turns on LEDs 0, 1, and 2. Here is a start:

```
void bar(unsigned int n) {
    if (n >= 1) P1OUT |= BIT0;   else P1OUT &= ~BIT0;
    if (n >= 2) P1OUT |= BIT1;   else P1OUT &= ~BIT1;
    ... complete code for LEDs 2 ... 7
}
```

At the start of `main`, declare a variable `i` as follows and configure all bits of P1 as outputs (leave the code for the watchdog timer alone) and write code that spins through the bar graph:

```
int main( void )
{
    int i = 0;

    // Stop watchdog timer to prevent time out reset
    WDTCTL = WDTPW + WDTTHOLD;

    P1OUT = 0;           // set P1.0 ... 7 low
    P1SEL = 0;          // configure P1.0 ... 7 for digital io
    P1DIR = 0xff;       // configure P1.0 ... 7 as outputs
```

```

for (;;) {
    bar(i);                // display
    i = i + 1;            // increase bar
    if (i > 8) i = 0;     // maximum length of bar is 8 LEDs on
    delay(50000);         // wait for humans to see result
}
}

```

Make sure all LED jumpers 0 ... 7 on J5 are placed. Run the program and marvel at the result.

3. Digital Input

Now will modify our code from the last section such that the bar graph advances one position each time a button is pressed. We will use P1.7 for the button input. *Remove the jumper for LED 7.*

The figure shows the circuit for connecting a button to the microcontroller. Normally P1.7 is pulled to VCC (high) by the pull-up resistor R_{up} . Pressing the button pulls P1.7 low. The circuit is even simpler than this since the microcontroller has a built-in R_{up} , we just need to enable it. So all you need to do is connect a button between P1.7 and ground.

Here is the code for enabling P1.0 ... 6 as outputs and P1.7 as an input with the pull-up resistor enabled:

```

P1OUT = BIT7;
P1SEL = 0;
P1DIR = 0x7f;
P1REN = BIT7;

```

The following statement checks for P1.7 to go low:

```

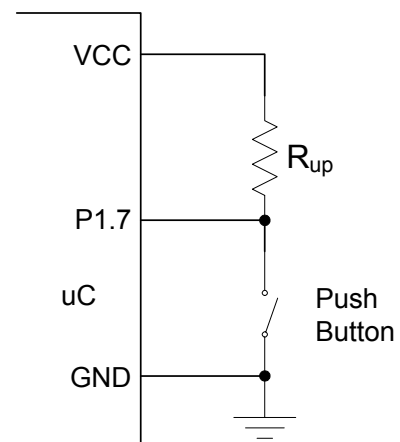
while (P1IN & BIT7) ;      // wait for P1.7 to go low

```

Put this into the infinite loop to control the bar graph display.

Start your program and verify first with the voltmeter that P1.7 is normally high and turns low when the button is pressed. When trying the program you will notice that things do not work as expected: when pressing the button the bar graph does not advance a single position as it should but a random number of positions. This has two reasons:

- Just like human eyes, human hands are pretty slow on a microcontroller timescale. Once the microcontroller has found that P1.7 is low and advanced the bar graph, it returns to the top of the loop, and checks again. If it finds the button still pressed, it advances the bar graph again.
- Most mechanical switches and buttons have a flaw called *prelling*: rather than just turning on and off, the button turns on and off rapidly in succession until it reaches the final value. The reason is mechanical resonance in the switch.



The first problem can be avoided by inserting a statement that checks that the button has gone back high into the loop:

```
while (!(P1IN & BIT7)) ;    // wait for P1.7 to go high
```

Depending on where exactly you insert the statement the bar graph advances immediately when the button is pushed or only when the button is released. Be sure that you are not trying to set P1.7 as an output in your bar subroutine, since P1.7 is the input used for the switch.

The solution for the second problem is inserting a delay into the loop. Fortunately we wrote that delay function!

Code for bar graph with button _____ of 30 **P**

Ask the lab TA to verify the bar graph and button: _____ of 10 **M**

SUGGESTIONS AND FEEDBACK

Time for completing prelab:

Time for completing lab:

Please explain difficulties you had and suggestions for improving this laboratory. Be specific, e.g. refer to paragraphs or figures in the write-up. Explain what experiments should be added, modified (how?), or dropped.

PRELAB SUMMARY

Lab Session:

Name 1:

SID:

Name 2:

SID:

Summarize your prelab (**P**) results here and turn this in at the beginning of the lab session.

Problem	Result
3	Attach code for part 3